ORACLE
FUSION MIDDLEWARE

An Oracle White Paper
April 2012

AIA 2.x Performance Tuning Show Case

ORACLE

Introduction 2
How This Tutorial is Structured 2
How to Use This Tutorial 2
Asynchronous Integration Flows 3
Asynchronous Test Scenario 3
Testing Methodology 4
Measuring Performance 5
ORA-12519 When Submitting Orders 5
Insight into Garbage Collection Behavior Required 6
Tuning JVM Memory Management 7
Performance Baseline for the Asynchronous Flow 12
DB Host Not Having Enough Memory 12
Tuning BPEL Configuration 13
Turning off Dehydration for Transient Processes 14
Adapter Tuning 16
Database Tuning 21
Implementation 22
Observation: Missing Indexes on X-Referencing Table 24
Synchronous Integration Flows 26
Synchronous Test Scenario 26
Testing Methodology 27
Measuring Performance 27
Baseline Test Results 28
Turning Off Dehydration 28
Missing Performance Patch 30
Turning off ESB Instance Tracking 31
The Impact of External Application Latency 32
Additional Tuning Strategies 34
Horizontal Scaling 34
Vertical Scaling 34
Specialized Deployment 35

Introduction

This document provides realistic scenarios showing how customers of Oracle AIA Process Integration Packs (PIPs), version 2.x can successfully approach the tuning of the middleware stack.

This tutorial can be seen as complementary to the white paper 'AIA 2.x Process Integration Packs – Performance Tuning'. The whitepaper covers all of the relevant areas of tuning that apply to a PIP deployment including infrastructure database tuning, engine (Oracle BPEL Manager / Oracle Enterprise Service Bus) tuning, JCA adapter tuning, and Java Virtual Machine (JVM) tuning.

In this tutorial you will see how to apply the tuning knobs in a real-world scenario. We use a non-tuned AIA deployment running the Communications Order-to-Bill PIP and Communications Agent Assisted Billing Care (AABC) PIP. We chose these two PIPs because they are popular and widely used and cover a wide variety of integration use cases.

While the Order-to-Bill PIP primarily implements asynchronous interaction patterns leveraging fire-and-forget and request-delayed-response integrations, the AABC PIP is basically on the other end of the spectrum providing synchronous integration flows. The very different nature of these integration flows can demand different tuning activities. There are also changes that apply to any integration flow regardless of type, for example, a well tuned JVM is a requirement for good overall performance in any case.

How This Tutorial is Structured

First we explain the problem statement in detail for each PIP that is to be tuned in this exercise. We explain the behavior of the system in a non-tuned configuration, show how we measure performance in our use case, and show the initial baseline results.

Then we incrementally approach each tuning area one by one. We highly recommend that you use a similar approach and only change one thing at a time and immediately verify how this improves or changes the system behavior under load. At every stage in the tuning exercise we go through a cycle consisting of observation, analysis, implementation, and result.

In the **Observation** section we show what we see in the system and how we made that information visible. The **Analysis** discusses the meaning of what we saw and based on the analysis, how to improve performance. **Implementation** shows how to translate the conclusions into configuration or code changes and finally the **Results** section visualizes the outcome in various ways.

How to Use This Tutorial

While you could pick certain points of interest from this tutorial and just read through these, we recommend that you read the guide from end-to-end to get the full context. Note that most of the tuning activities you find in this guide can apply both to synchronous and asynchronous integration flows, though they might only appear in one context. This guide does not try to be complete in a sense

that it covers all possible things you could do. However it does document the tuning that proved to be useful in the concrete example.

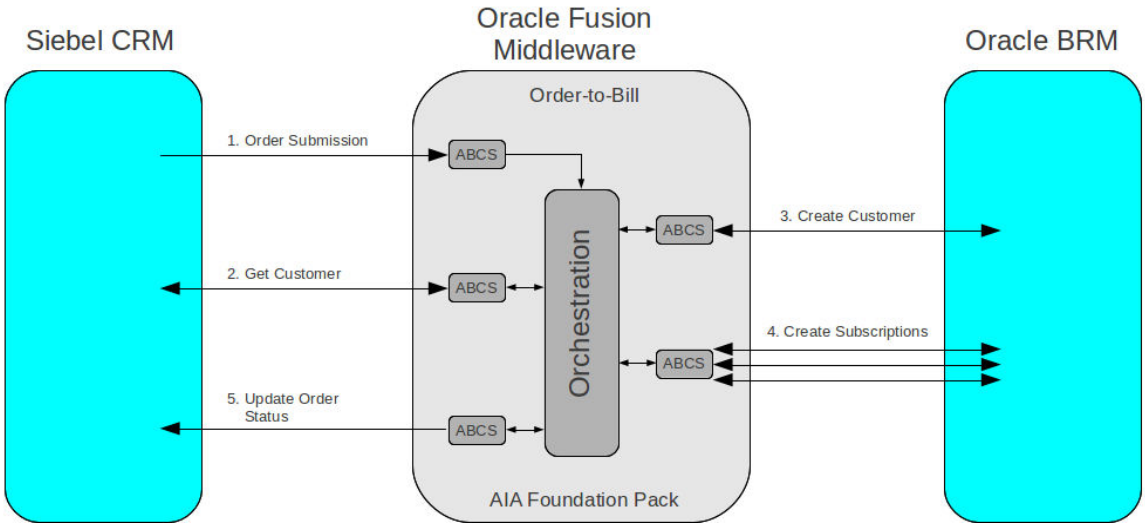
In the last section we discuss some additional tuning strategies that were not covered in the tutorial.

Asynchronous Integration Flows

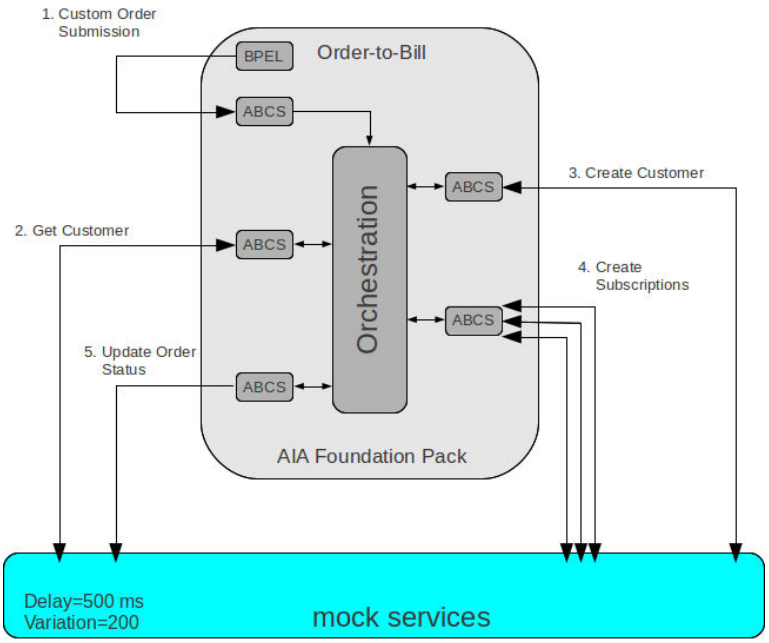
Asynchronous Test Scenario

The AIA Order-to-Bill PIP is a great example of a complex, mostly asynchronous integration scenario. The core of this PIP is an implementation of an order orchestration process in the context of a customer in the Communications industry using Oracle Siebel as the CRM system and Oracle Billing and Revenue Management (BRM) for customer billing purposes.

The following diagram shows the order submission and orchestration in the delivered AIA Order-to-Bill PIP.



The diagram illustrates a typical asynchronous flow for order processing. Siebel puts a message for a new order for a new customer into a specific Oracle Advanced Queue (AQ) (step 1). The AIA PIP picks up the message from there and feeds the transformed order payload into the orchestration service which, in our case. is a BPEL process. This orchestration service leverages various other AIA services to get customer details (2) from Siebel (which were not part of the original order payload), then creates the customer in BRM (3), subsequently creates the subscriptions for this new account in BRM (4) through multiple API calls and finally updates Siebel in order to reflect the outcome of the order processing (5).



In order to make performance tests reproducible in such a scenario and also to have the ability to remove dependencies on applications, we replace both Siebel and BRM with mock services simulating their APIs. Also, we create a custom order submission tool (a custom BPEL process) that generates the load for the tests in the initial queue:

This setup enables the analysis and tuning of the middleware components in isolation.

Testing Methodology

The main reason for initiating tests for asynchronous order processing is to enqueue a batch of new orders into the same queue that Siebel would typically use to place orders that will be picked up by the AIA PIP. The custom BPEL process is able to generate any number of unique orders based on a payload template. We used the following testing standards:

- With templates for small, medium, and large orders we can simulate different situations. For our tests, we work with batches of 500 orders. This ensures that tests run long enough (for at least one hour initially) to rule out random events and results become comparable.
- We always 'warm up' the environment ahead of starting to generate a steady-state condition. We do not want to measure the cost of bringing all required components into memory, eventually compile or initialize things, and so on. Though that might be interesting in a different context, it is not relevant for us in this use case.
- We make sure to purge any data from the database that was generated in earlier test runs to ensure test results are comparable. This purging includes the persistence tables for BPEL and ESB (refer to

the documentation on how to purge those) as well as cleaning up the X-Referencing table from the previous run as well.

Since the X-Referencing table is a hotspot, we do not want it to keep growing because that would slow down later tests. We remove all entries from the table that were generated after a certain point in time to leave the setup related entries (for example, product mappings required for the flow) while removing transactional entries: ***delete from aia.xref_data where last_modified > 1322738435969***.

- We make sure that all tablespaces in the database have enough space for all data that will be generated during tests. It can be difficult to fix the environment if your databases are running out of space during performance tests.
- Before running tests we update the database statistics in order to give the database a chance to perform optimally. Although there are different ways to do this we run the PL/SQL procedure DBMS_STATS.GATHER_DATABASE_STATS to update statistics for the entire database.

Measuring Performance

It is essential to understand the current performance of the system before doing any performance tuning exercise. Otherwise it becomes difficult, or even impossible, to understand if certain changes are beneficial or not. In addition there must be a clear understanding on how to measure performance.

- For the asynchronous order execution, we use the BPEL internal persistence in the database for evaluating the system performance.
  - By using the table CUBE_INSTANCE in the BPEL database schema we can tell when a test started, when it ended, and how many orders were processed. Since the CUBE_INSTANCE table has one record including timestamps for every single BPEL instance as long as the service is not configured as transient, this is the ideal source to understand what the SOA infrastructure processed and when.
- We default these values from the first successful test runs to be 100 and from then on only measure and visualize the difference to these baseline values going forward in the tuning exercise.

ORA-12519 When Submitting Orders

Observation

After submitting the first set of orders, we look at the BPEL and ESB consoles and track the JVM output on the middleware server. We can see that the order cannot be processed successfully due to ORA-12519 failures.

Analysis

From the error code we conclude that the database apparently runs out of connections under a heavier load on the middleware. We did not see this behavior when processing single orders; it appears only when we submit batches of orders. The conclusion is that more concurrent processes on the SOA

engines ask for more sessions from the infrastructure database than it is configured for. Therefore, the number of sessions must be increased.

Implementation

Changing the number of sessions along with the number of processes is a standard Oracle database maintenance activity and can be performed with the following SQL commands:

```
ALTER SYSTEM SET PROCESSES=1000 SCOPE=SPFILE;

ALTER SYSTEM SET SESSIONS=1200 SCOPE=SPFILE;
```

Note: These changes only take effect after restarting the Oracle database.

Insight into Garbage Collection Behavior Required

Observation

When we look at the SOA server logs and the BPEL console, we see that services fail while trying to execute on a batch of orders. When we review the SOA server log more closely, it turns out that the root cause for failures are out-of-memory exceptions happening when we create a significant load by submitting a few hundred orders at once.

Analysis

Simply increasing the memory of the Java Virtual Machine (JVM) is not necessarily the right thing to do even if it might help to work around a particular problem. To assess the situation, we need insight into JVM's Garbage Collection (GC) behavior to understand how memory is being utilized in the JVM.

Implementation

All JVM implementations provide logging capabilities for memory consumption and memory management events. This can be visualized by adding GC debugging switches in the central Oracle Application Server 10g configuration file opmn.xml. The relevant switches for the JVM hosting BPEL and ESB are: ***-XX:+PrintGCTimeStamps -XX:+PrintGCDetails -verbose:gc***.

Results

By reviewing the SOA server log w can now see when memory cleanup events occur, how long they take to complete, and how much memory they free up:

```
855.203: [Full GC [PSYoungGen: 711457K->199665K(1046144K)] [PSOldGen:
729688K->730664K(731136K)] 1441145K->930329K(1777280K) [PSPermGen: 171665K-
>171142K(346112K)], 2.0546950 secs]
858.967: [Full GC [PSYoungGen: 711808K->308427K(1046144K)] [PSOldGen:
730664K->730664K(731136K)] 1442472K->1039091K(1777280K) [PSPermGen: 171145K-
>171145K(331776K)], 1.5443340 secs]
862.622: [Full GC [PSYoungGen: 711597K->363800K(1046144K)] [PSOldGen:
730664K->730664K(731136K)] 1442262K->1094465K(1777280K) [PSPermGen: 171155K-
>171155K(317440K)], 1.7730850 secs]
```

Tuning JVM Memory Management

Observation

Once we can visualize the garbage collection (GC) behavior of the JVM, we can see what happens for a batch of a few large orders. Essentially the JVM is busy trying to free up memory byrunning 'full GCs':

```
855.203: [Full GC [PSYoungGen: 711457K->199665K(1046144K)] [PSOldGen:
729688K->730664K(731136K)] 1441145K->930329K(1777280K) [PSPermGen: 171665K-
>171142K(346112K)], 2.0546950 secs]
858.967: [Full GC [PSYoungGen: 711808K->308427K(1046144K)] [PSOldGen:
730664K->730664K(731136K)] 1442472K->1039091K(1777280K) [PSPermGen: 171145K-
>171145K(331776K)], 1.5443340 secs]
862.622: [Full GC [PSYoungGen: 711597K->363800K(1046144K)] [PSOldGen:
730664K->730664K(731136K)] 1442262K->1094465K(1777280K) [PSPermGen: 171155K-
>171155K(317440K)], 1.7730850 secs]
868.859: [Full GC [PSYoungGen: 711274K->391376K(1046144K)] [PSOldGen:
730664K->730664K(731136K)] 1441939K->1122040K(1777280K) [PSPermGen: 171197K-
>171197K(301056K)], 1.5306590 secs]
875.493: [Full GC [PSYoungGen: 710471K->354880K(1046144K)] [PSOldGen:
730664K->729592K(731136K)] 1441135K->1084472K(1777280K) [PSPermGen: 171199K-
>171175K(286720K)], 1.8621180 secs]
883.438: [Full GC [PSYoungGen: 710724K->386294K(1046144K)] [PSOldGen:
729592K->729592K(731136K)] 1440316K->1115886K(1777280K) [PSPermGen: 171175K-
>171175K(272384K)], 2.0807750 secs]
```

Analysis

The values for PSOldGen show that even the full GC cycles cannot free up any memory in the old generation (example: PSOldGen: 729592K->729592K). Since there are no other options the JVM could choose, the JVM keeps running costly full GCs ending up in a 'constant full GC mode'. This is because there are long-running processes requiring a lot of memory, and all the objects in the memory are effectively in use and cannot be removed from memory.

To make use of the latest JVM capabilities and tuning options, we replace the JDK 1.5 (1.5.0_22 in our case) underneath SOA with the latest JDK 1.6 (1.6.0_27).

Implementation

We download the most recent JDK 1.6 implementation (from otn.oracle.com for most operating systems), extract the content into a directory on the server and then replace <ORACLE_HOME>/jdk with the extracted directory from the download.

Results

Using JDK 1.6 we can change -XX:+PrintGCTimeStamps to ***-XX:+PrintGCDateStamps*** to see proper time stamps in the GC logs as opposed to only seeing the time offset since the JVM start.

Running another set of tests on JDK 1.6 with large payloads shows this GC behavior:

```
2011-11-22T02:14:21.040-0800: [Full GC [PSYoungGen: 132651K-
>24914K(1099008K)] [PSOldGen: 490964K->595282K(731136K)] 623615K-
```

```
>620196K(1830144K) [PSPermGen: 178545K->178545K(401408K)], 1.7995110 secs]
[Times: user=1.73 sys=0.06, real=1.80 secs]
2011-11-22T02:14:26.061-0800: [Full GC [PSYoungGen: 831303K-
>13877K(1099008K)] [PSOldGen: 595282K->729776K(731136K)] 1426585K-
>743653K(1830144K) [PSPermGen: 178547K->178547K(387072K)], 2.0478080 secs]
[Times: user=1.92 sys=0.14, real=2.05 secs]
2011-11-22T02:14:30.712-0800: [Full GC [PSYoungGen: 830895K-
>88591K(1099008K)] [PSOldGen: 729776K->730814K(731136K)] 1560672K-
>819405K(1830144K) [PSPermGen: 178549K->178175K(368640K)], 2.2517120 secs]
[Times: user=2.25 sys=0.01, real=2.25 secs]
2011-11-22T02:14:47.009-0800: [Full GC [PSYoungGen: 831598K-
>234383K(1099008K)] [PSOldGen: 730814K->730814K(731136K)] 1562413K-
>965198K(1830144K) [PSPermGen: 178196K->178196K(352256K)], 1.7808800 secs]
[Times: user=1.78 sys=0.00, real=1.78 secs]
2011-11-22T02:14:55.945-0800: [Full GC [PSYoungGen: 831285K-
>278607K(1099008K)] [PSOldGen: 730814K->730814K(731136K)] 1562099K-
>1009421K(1830144K) [PSPermGen: 178237K->178237K(335872K)], 2.0979480 secs]
[Times: user=2.10 sys=0.01, real=2.10 secs]
```

Analysis

The GC behavior has not changed with the uptake of the latest 1.6 JDK since the JVM quickly runs out of tenured space. We need to remove the constraint of not having enough memory available to the JVM. In this scenario, we can either increase the overall heap, increase only the tenure area, or both.

The initial memory configuration of the JVM hosting BPEL and ESB runtime looks like this for our test case: -mx2048M -ms1560M -XX:MaxPermSize=512M -XX:+UseLargePages -XX:+AggressiveHeap -XX:MaxNewSize=1334M -XX:NewSize=1334M -XX:AppendRatio=3.

Implementation

We decide to change the basic amount of memory available to the JVM and to define other memory management related settings:

- We set '-ms' to the same value as '-mx' to avoid additional full GCs due to re-sizing the heap. This is generally a best practice.
- The ratio of young vs. tenure space is currently set to 1334 / 2048 = 65%. This is a reasonable value for typical AIA configurations. However, since we have seen before that we do not have enough tenure space, we set this ratio to 50% for the next test.
- MaxPermGen is currently set to 512MB. Earlier runs indicate that we do not need that much memory (e.g. PSPermGen: 178237K->178237K(335872K)), so we set it to 256MB in order to free up some memory for use in the heap. Note that this value might not be sufficient in other deployments where more BPEL processes are running than in this example.
- We set the initial size via PermGen to the same value as MaxPermGen to avoid any costly resizing of the PermGen space at runtime.
- We add '-XX:SurvivorRatio=8' to designate 1/8th of the young space as survivor space (default 1/32th)

- We remove '-XX:+AggressiveHeap' and add '-XX:-UseAdaptiveSizePolicy' instead. Note that removing '-XX:+AggressiveHeap' alone would not have any effect since this is the default setting starting with JDK 1.6. With these two changes, we disable the JDK ergonomics features and do not allow the JDK to dynamically resize the survivor space based on runtime statistics. This means that we can maintain the intended value of 1/8th as defined by the '-XX:SurvivorRatio=8' switch.
- We remove -XX:+UseLargePages since it is not used at the platform for this test.
- We remove the deprecated option -XX:AppendRatio=3.
- We replace both '-XX:NewSize=1334M -XX:MaxNewSize=1334M' with the recommended switch for sizing the '-Xmn1126', that is, set the initial and max size for the young generation to a value of 1126M.

Results

This results in the following memory settings: ***-ms2048M -mx2048M -Xmn1126M -XX:PermSize=256M -XX:MaxPermSize=256M -XX:SurvivorRatio=8 -XX:-UseAdaptiveSizePolicy***.

```
2011-11-22T02:26:59.501-0800: [Full GC [PSYoungGen: 761472K-
>761472K(955648K)] [PSOldGen: 942892K->942892K(944128K)] 1704364K-
>1704364K(1899776K) [PSPermGen: 178091K->178091K(262144K)], 1.7065060 secs]
[Times: user=1.71 sys=0.00, real=1.71 secs]
2011-11-22T02:27:01.216-0800: [Full GC [PSYoungGen: 761472K-
>761472K(955648K)] [PSOldGen: 942893K->942893K(944128K)] 1704365K-
>1704365K(1899776K) [PSPermGen: 178091K->178091K(262144K)], 1.6076450 secs]
[Times: user=1.61 sys=0.00, real=1.61 secs]
2011-11-22T02:27:02.825-0800: [Full GC [PSYoungGen: 761472K-
>761472K(955648K)] [PSOldGen: 942893K->942893K(944128K)] 1704365K-
>1704365K(1899776K) [PSPermGen: 178091K->178091K(262144K)], 1.6433230 secs]
[Times: user=1.65 sys=0.00, real=1.64 secs]
2011-11-22T02:27:04.470-0800: [Full GC [PSYoungGen: 761472K-
>748011K(955648K)] [PSOldGen: 942893K->942636K(944128K)] 1704365K-
>1690648K(1899776K) [PSPermGen: 178091K->178091K(262144K)], 1.9526180 secs]
[Times: user=1.95 sys=0.01, real=1.96 secs]
```

At the same time, we see the following in the SOA server log:

```
11/11/22 02:27:10 java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Analysis / Implementation

With JDK 1.6, GC takes most of the available CPU resources and the application cannot make any progress. Therefore, it is time to increase the heap space to 4GB. This results in the following settings: ***-ms4096M -mx4096M -Xmn2252M -XX:PermSize=256M -XX:MaxPermSize=256M -XX:SurvivorRatio=8 -XX:-UseAdaptiveSizePolicy***.

Results

Even with 4GB of heap we observe constant full GC cycles as well as the same GC overhead limit message as before:

```
2011-11-22T05:49:15.681-0800: [Full GC [PSYoungGen: 1514445K-
>1389952K(1897344K)] [PSOldGen: 1888167K->1888167K(1888256K)] 3402612K-
>3278119K(3785600K) [PSPermGen: 179555K->179555K(262144K)], 2.2387780 secs]
[Times: user=2.28 sys=0.00, real=2.24 secs]
2011-11-22T05:49:18.202-0800: [Full GC [PSYoungGen: 1515648K-
>1410675K(1897344K)] [PSOldGen: 1888167K->1888167K(1888256K)] 3403815K-
>3298842K(3785600K) [PSPermGen: 179555K->179555K(262144K)], 1.9773680 secs]
[Times: user=2.01 sys=0.00, real=1.97 secs]
2011-11-22T05:49:20.343-0800: [Full GC [PSYoungGen: 1515392K-
>1426968K(1897344K)] [PSOldGen: 1888167K->1888167K(1888256K)] 3403559K-
>3315135K(3785600K) [PSPermGen: 179555K->179555K(262144K)], 2.0455250 secs]
[Times: user=2.08 sys=0.00, real=2.04 secs]
2011-11-22T05:49:22.582-0800: [Full GC [PSYoungGen: 1513947K-
>1364438K(1897344K)] [PSOldGen: 1888167K->1887481K(1888256K)] 3402114K-
>3251919K(3785600K) [PSPermGen: 179555K->179555K(262144K)], 3.1434140 secs]
[Times: user=3.19 sys=0.00, real=3.14 secs]
...
11/11/22 05:51:48 java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Analysis

Moving to 4GB has not significantly improved the situation in the old generation, so we keep adding more head space to the JVM and now move to 6GB: ***-ms6144M -mx6144M -Xmn2468M -XX:PermSize=256M -XX:MaxPermSize=256M -XX:SurvivorRatio=8 -XX:-UseAdaptiveSizePolicy***. Old Gen is increasing even within Full GC cycles. Therefore, we want to use the parallel old GC algorithm (default is single-threaded collection).

Note that the ratio of young space of the overall heap is now about 40%. This is a low percentage compared to other well performing SOA deployments where best performance is seen with a higher young ratio of 50-65%. But AIA deployments running mostly asynchronous flows have a special memory consumption pattern where a lower young generation is better.

Implementation

This can be done by adding the following parameters: ***-XX:+UseParallelOldGC -XX:ParallelGCThreads=4*** (note that we have 4 CPUs on this box with active JVM, hence, providing all CPUs to the GC collection is okay here).

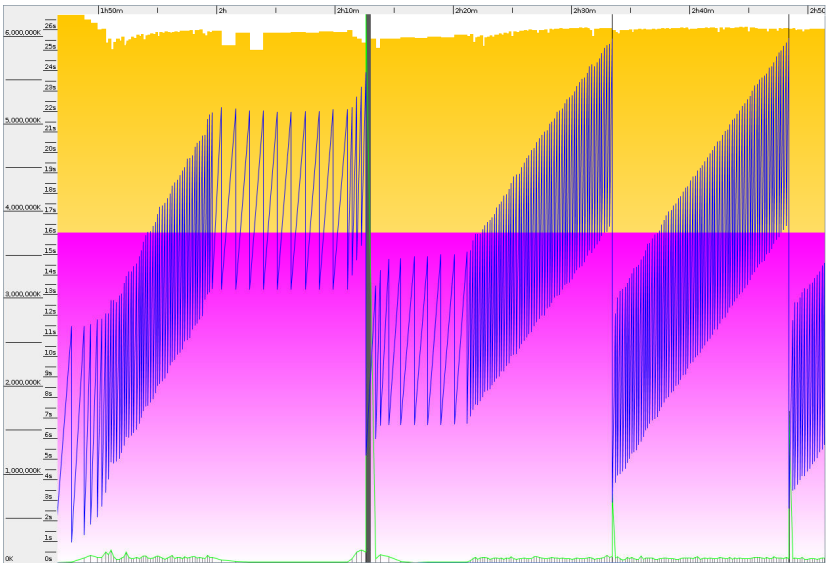
Also, we add the switch ***-Dcom.sun.management.jmxremote=true*** as we want to allow the JConsole tool access to JVM. This enables us to use JConsole to get further online insight into the JVM if required.

Results

Finally, large orders start to complete successfully, but we still see full GC phases (but with reasonable application time in between). The JVM later 'recovers' from this scenario after most long-running xml/xsl intensive tasks are done and objects residing in the tenured area can be released.

```
2011-11-22T08:32:52.327-0800: [Full GC [PSYoungGen: 235290K->0K(2116224K)]
[ParOldGen: 3434685K->3619481K(3764224K)] 3669975K->3619481K(5880448K)
[PSPermGen: 181509K->181132K(524288K)], 12.1453960 secs] [Times: user=22.35
sys=2.67, real=12.14 secs]
2011-11-22T08:33:48.347-0800: [Full GC [PSYoungGen: 1689989K->0K(2116224K)]
[ParOldGen: 3621070K->3674779K(3764224K)] 5311060K->3674779K(5880448K)
[PSPermGen: 181181K->181159K(524288K)], 4.4352350 secs] [Times: user=10.23
sys=0.78, real=4.44 secs]
2011-11-22T08:34:06.925-0800: [Full GC [PSYoungGen: 1690003K->0K(2116224K)]
[ParOldGen: 3676369K->3675617K(3764224K)] 5366372K->3675617K(5880448K)
[PSPermGen: 181164K->181161K(524288K)], 4.4027110 secs] [Times: user=12.88
sys=0.01, real=4.40 secs]
2011-11-22T08:34:45.552-0800: [Full GC [PSYoungGen: 1690335K->0K(2116224K)]
[ParOldGen: 3675617K->403644K(3764224K)] 5365952K->403644K(5880448K)
[PSPermGen: 181280K->181275K(524288K)], 3.2167610 secs] [Times: user=6.28
sys=0.00, real=3.21 secs]
2011-11-22T08:35:21.056-0800: [GC [PSYoungGen: 1690256K->95503K(2101376K)]
2093900K->499147K(5865600K), 0.0846200 secs] [Times: user=0.32 sys=0.00,
real=0.08 secs]
2011-11-22T08:36:35.341-0800: [GC [PSYoungGen: 1785059K->142211K(2123008K)]
2188704K->598314K(5887232K), 0.1492370 secs] [Times: user=0.58 sys=0.00,
real=0.15 secs]
2011-11-22T08:36:54.850-0800: [GC [PSYoungGen: 1851660K->116999K(2113792K)]
2307763K->686333K(5878016K), 0.1823910 secs] [Times: user=0.62 sys=0.00,
real=0.18 secs]
```

In many cases it makes sense to visualize GC log information. There are various tools available for that. 'GCViewer' is used here to plot this diagram:



This diagram indicates full GC cycles (black bars) as well as many minor collections taking place. Overall this generates the desired 'saw tooth pattern' which is usually considered the ideal behavior. This means that minor collections are able to free up significant portions (>95% in our example) of the Eden Space while full GC cycles (that typically stop the whole JVM for the duration of the collection!) only happen from time to time (every 15 to 20 minutes in the diagram above). It is also important to note that major collections also free up significant space (e.g. ~3.5GB in the diagram above).

Performance Baseline for the Asynchronous Flow

Now, that we have stabilized the environment, we are successfully able to run tests for all payload sizes in significant batch sizes. The initial numbers look like this, depending on the payload:

ORDER SIZE	THROUGHPUT (ORDERS PER MINUTE)
Small	6.67
Medium	3.02
Large	1.74

In the Order-to-Bill PIP, the order size apparently has significant impact on the throughput including the need to handle larger messages in memory for larger orders, and the need to have many more API calls towards the billing system for larger orders as compared to smaller orders.

We default these values to 100 and only measure and visualize the difference to this baseline value going forward in the tuning.

DB Host Not Having Enough Memory

Observation

To understand the behavior of the infrastructure DB when processing orders we run an ADDM report (see the Oracle Database documentation for details on how to generate an ADDM report with these results:

```
        DETAILED ADDM REPORT FOR TASK 'TASK_214' WITH ID 214
           ----------------------------------------------------

              Analysis Period: 24-NOV-2011 from 01:20:42 to 02:02:39
            Database ID/Instance: 1293781610/1
          Database/Instance Names: ORCL/orcl
                       Host Name: db.aiaperf.com
                Database Version: 10.2.0.5.0
                  Snapshot Range: from 96 to 98
                    Database Time: 8159 seconds
             Average Database Load: 3.2 active sessions

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

FINDING 1: 100% impact (8159 seconds)
```

```
-------------------------------------
Significant virtual memory paging was detected on the host operating system.

   RECOMMENDATION 1: Host Configuration, 100% benefit (8159 seconds)
      ACTION: Host operating system was experiencing significant paging but no
         particular root cause could be detected. Investigate processes that
         do not belong to this instance running on the host that are consuming
         significant amount of virtual memory. Also consider adding more
         physical memory to the host.
```

Using the OS level top tool confirm that the operating system makes use of swap space:

```
top - 04:16:02 up  3:45,  2 users,  load average: 0.05, 0.02, 0.00
Tasks: 173 total,   1 running, 172 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.3%us,  2.6%sy,  0.0%ni, 94.9%id,  0.0%wa,  0.2%hi,  0.0%si,  0.0%st
Mem:   3047856k total,  2348320k used,   699536k free,    35788k buffers
Swap:  2097144k total,   206504k used,  1890640k free,  1991376k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2608 oracle    20   0 1319m 845m 843m S  0.0 28.4   0:30.53 oracle
 2606 oracle    20   0 1315m 131m 131m S  0.0  4.4   0:01.88 oracle
 2787 oracle    20   0 1316m  95m  93m S  0.0  3.2   0:05.14 oracle
```

Analysis

The database host does not have enough memory assigned, so it starts swapping, that is, using the disk to extend the memory since the available physical RAM is not sufficient. This must be resolved either by providing more memory to the box or adjusting the Oracle memory configuration. In our case we increase the memory.

Implementation

Since we are working on a virtualized environment, we simply assign more memory to the virtual machine (VM) hosting the database (3.0 to 3.5 GB in our example).

Tuning BPEL Configuration

Observation

Tuning the BPEL engine is a typical task that should always be done regardless of the use case. There are no observations as such, but you use the BPEL console to verify and change most of the BPEL engine configuration parameters.

Analysis

To verify and change the BPEL Domain-level configuration in the BPEL, we change auditLevel to production:

ORACLE Enterprise Manager 10g
BPEL Control

Dashboard | Processes | Instances | Activities

BPEL Domain: default
Statistics: 74 Active Processes | 0 Retired Processes
Build version: 10.1.3.5.0 [build #0] - type: release

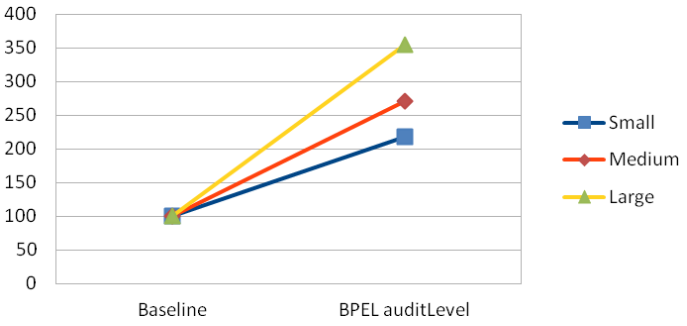
Domain | XPath Library | Logging | Auto-Recovery | Resources

Configuration Descriptor of this BPEL Domain

Property	Value	Name
auditDetailThreshold	50000	Audit trail details logging threshold
auditLevel	production	Audit trail logging level
bpelcClasspath	/home/oracle/soa/bpel/system/classes:/home/oracle/soa/bpel/lib/j2e	BPEL process compiler classpath
completionPersistLevel	all	Completion persist level of transient processes
completionPersistPolicy	faulted	Completion persist policy of transient processes
datasourceJndi	jdbc/BPELServerDataSourceWorkflow	Domain datasource JNDI
deliveryPersistPolicy	on	persist delivery messages
dspEngineThreads	60	Engine thread pool size
dspInvokeThreads	40	Invoke thread pool size
dspMaxRequestDepth	2000	dspMaxRequestDepth
dspSystemThreads	2	System thread pool size
expirationMaxRetry	5	Maximum expiration retry limit
instanceKeyBlockSize	10000	instance key block size
largeDocumentThreshold	50000	Large XML document persistence threshold

Results

With this change, we see significant improvement compared to the initial results since the overhead for tracking audit trail information for the BPEL processes is much lower now:



Turning off Dehydration for Transient Processes

Observation

There is heavy DB usage during order processing as shown using mpstat -P all output on DB:

```
06:20:39 AM  CPU    %user    %nice     %sys %iowait     %irq   %soft  %steal   %idle    intr/s
06:20:44 AM  all    11.50     0.00    18.70   51.30     4.30    6.00    0.00    8.20   3837.83
06:20:44 AM    0    12.07     0.00    20.45   59.92     1.23    1.64    0.00    4.70    254.81
06:20:44 AM    1    10.94     0.00    16.99   43.16     7.23    9.96    0.00   11.72   2177.71
```

```
06:20:44 AM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal   %idle    intr/s
06:20:49 AM  all   12.30    0.00   21.88   38.28    4.59    5.86    0.00   17.09   4542.94
06:20:49 AM    0   13.71    0.00   21.37   35.69    1.61    1.61    0.00   26.01    271.98
06:20:49 AM    1   10.84    0.00   22.43   40.68    7.60    9.89    0.00    8.56   2657.86

06:20:49 AM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal   %idle    intr/s
06:20:54 AM  all   12.76    0.00   23.05   36.60    4.65    6.92    0.00   16.02   4525.56
06:20:54 AM    0   13.79    0.00   26.17   40.77    1.42    2.03    0.00   15.82    292.09
06:20:54 AM    1   11.95    0.00   20.04   32.56    7.51   11.75    0.00   16.18   2581.74
```

Also, the AWR and ADDM report indicates contention on the cube_instance table:

```
FINDING 1: 55% impact (4474 seconds)
-----------------------------------
SQL statements consuming significant database time were found.

   RECOMMENDATION 1: SQL Tuning, 53% benefit (4357 seconds)
      ACTION: Investigate the SQL statement with SQL_ID "dv7aptbr6m6bn" for
         possible performance improvements.
         RELEVANT OBJECT: SQL statement with SQL_ID dv7aptbr6m6bn
         INSERT INTO audit_details ( cikey, domain_ref, detail_id, bin_csize,
         bin_usize, bin, ci_partition_date ) VALUES ( :1, :2, :3, :4, :5, :6,
         :7 )
```

Analysis

In our order processing scenario, most of the BPEL processes involved are transient in nature and, therefore, can be configured for pure memory processing avoiding any backend interaction. Transient means that they do not have any dehydration point in their flows as it would be enforced by activities such as mid-process receive, wait, on-alarm. Transient services can be configured so they are handled purely in memory and are only persisted into the BPEL database tables if the execution fails.

Implementation

Configuring transient services can be done by editing the respective bpel.xml file and redeploying the service:

```
<BPELSuitcase>
    ...
    <configurations>
      ...
      <property name="inMemoryOptimization">true</property>
      <property name="completionPersistPolicy">faulted</property>
    </configurations>
  </BPELProcess>
</BPELSuitcase>
```

For the AIA Order to Bill PIP, we apply this to the following services:

- ProcessFOBillingAccountListRespOSMCFSCommsJMSProducer
- ProcessFulfillmentOrderBillingBRMCommsAddSubProcess
- ProcessFulfillmentOrderBillingBRMCommsProvABCSImpl

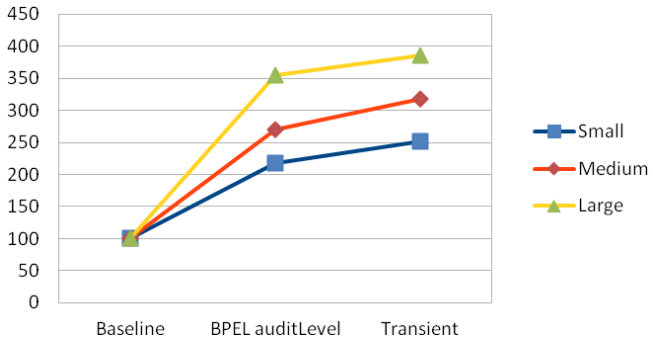
- ProcessFulfillmentOrderBillingResponseOSMCFSCommsJMSProducer
- ProcessSalesOrderFulfillmentOSMCFSCommsJMSProducer
- ProcessSalesOrderFulfillmentSiebelCommsReqABCSImpl
- QueryCustomerPartyListSiebelProvABCSImplV2
- SyncCustomerPartyListBRMCommsProvABCSImpl
- UpdateSalesOrderSiebelCommsProvABCSImpl

Note that list only covers the sample test case. The Order-to-Bill PIP supports many more flows, so in an actual customer deployment the list of relevant services for this kind of tuning would probably include several more services.

Transient services do not leave any trace in the system in such a configuration. While this is desirable from a performance point of view since it takes the burden from the infrastructure database, it might also introduce new challenges when it comes to error handling. These services show up in the consoles when they fail, however they are not visible when the caller of such a service fails. In such a situation, the called service would not create a visible instance (since it completed without an issue), but for the larger flow it might be helpful or even required to understand what the called service actually did., You will have to balance performance needs against tracking requirements.

Results

This leads to the following processing results:



The change to transient BPEL processes further decreases the overhead of storing instance data in the BPEL dehydration database leading to an improvement of about another 40%.

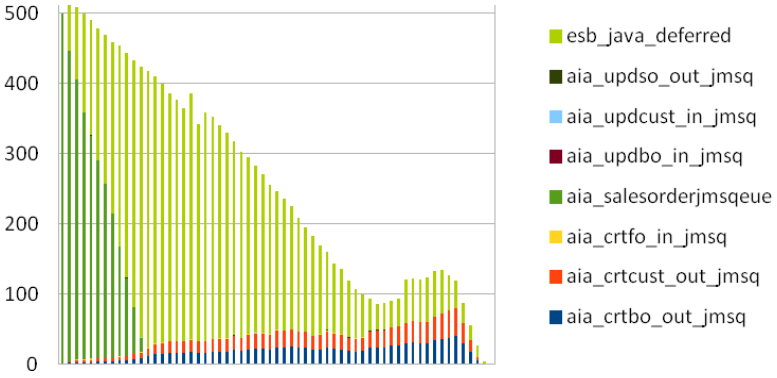
Adapter Tuning

Observation

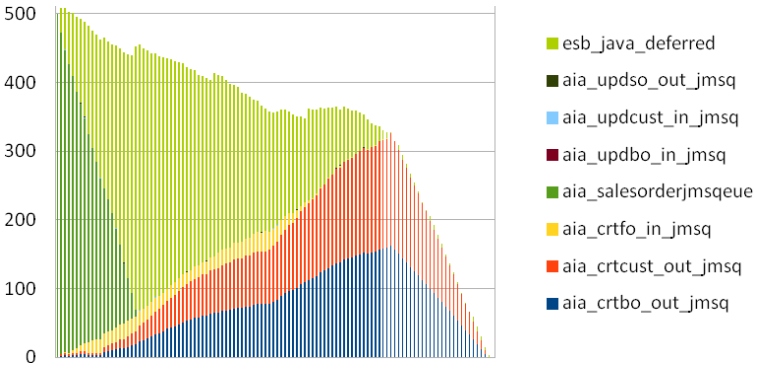
While the middleware CPU is under-utilized at times with the current configuration, it appears that there are enough orders waiting in some intermediate queues for further processing.

It helps to visualize where orders currently 'sit' in the overall processing. For this purpose, we use a simple SQL script to count ready messages in all relevant queues (e.g. 'select state, count(*) from jmsuser.aia_salesorderjmsqtab group by state') in constant intervals and then build a chart from that (x-axis are points in time, y-axis number of waiting messages in each queue):

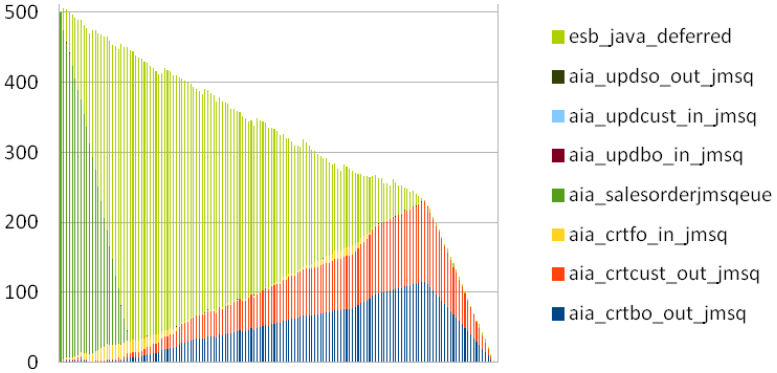
Small orders:



Medium orders:



Large orders:



All these queues are internal queues used in the AIA PIP except esb_java_deferred which is a system queue holding all messages that are waiting to be processed asynchronously in ESB.

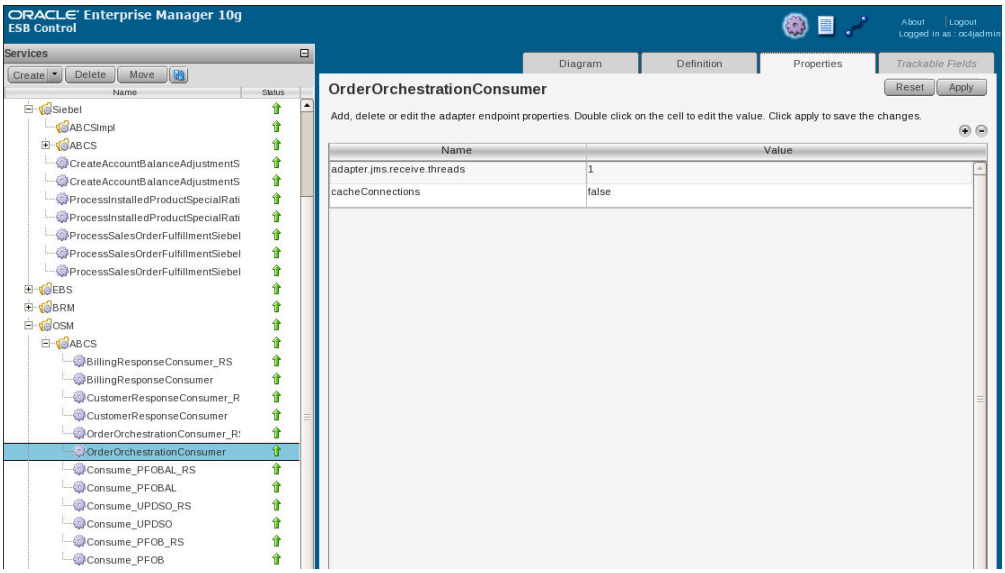
Another observation from the server output log is that the order processing appears to be sequential, at least in certain phases:

```
11/11/24 06:20:21 loaded from db
11/11/24 06:20:21 No Input
11/11/24 06:20:21  contactNameText
11/11/24 06:20:21 No Input
11/11/24 06:20:22 loaded from db
11/11/24 06:20:22 Siebel contact contactNameText
11/11/24 06:20:25 ---------------------------------------->>>>>>>>routeToCAVS true
11/11/24 06:20:31 loaded from db
11/11/24 06:20:31 No Input
11/11/24 06:20:31  contactNameText
11/11/24 06:20:31 No Input
11/11/24 06:20:32 loaded from db
11/11/24 06:20:32 Siebel contact contactNameText
11/11/24 06:20:35 ---------------------------------------->>>>>>>>routeToCAVS true
11/11/24 06:20:40 loaded from db
11/11/24 06:20:40 No Input
11/11/24 06:20:40  contactNameText
11/11/24 06:20:40 No Input
11/11/24 06:20:42 loaded from db
11/11/24 06:20:42 Siebel contact contactNameText
11/11/24 06:20:44 ---------------------------------------->>>>>>>>routeToCAVS true
```

Analysis

Because the default settings from JMS consumer services in ESB are defined as single-threaded adapters and the EBS-BPEL and BPEL-ESB invocations are configured to happen within the same transaction and therefore in the same thread, only one thread picks up messages from a queue and then processes it downstream to the next persistence point, usually to the next queue.

This behavior can be changed by increasing the adapter.jms.receive.threads setting per service in the ESB console:



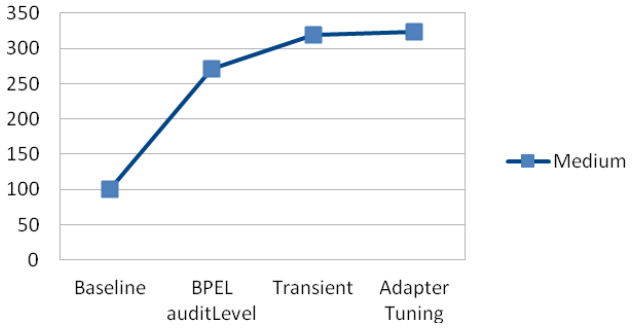
Implementation

We initially increase the number of threads per queue consumer as shown below:

QUEUE	CONSUMER SERVICE	ADAPTER.JMS.RECEIVE.THREADS
aia_crtcust_out_jmsq	Consume_PFOBAL	1 → 10
aia_crtbo_out_qtab	Consume_PFOB	1 → 10

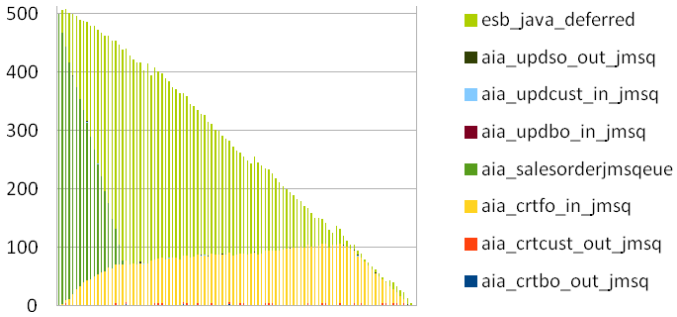
Results

The test results now look like this for medium orders:

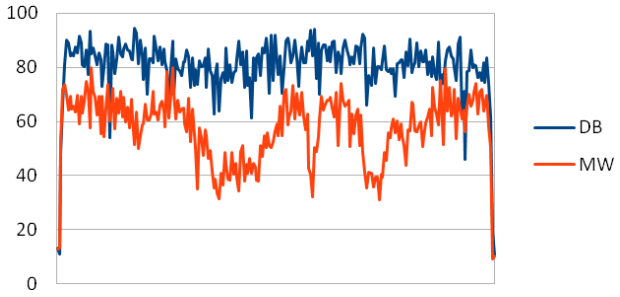


Note: We focus only on medium orders since testing with different payload sizes does not give us additional information at this point. But later on, we would certainly test again with all payload sizes to verify stability and performance independent of the payload.

With this change, we see a different queuing behavior.



This time, we also gather CPU load both on DB and middleware tier. This can be done (depending on the operating system) with tools such as mpstat:



Analysis

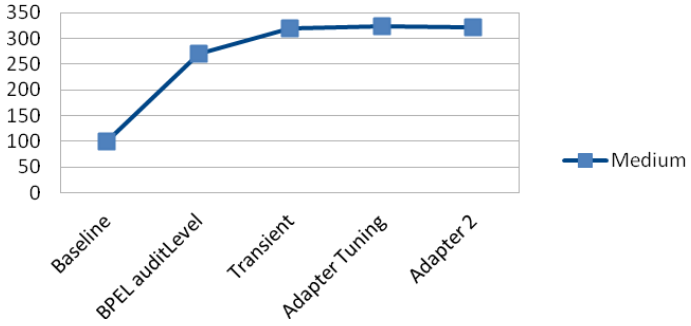
The change to the adapter shows immediate effects, but also delegates the issue to another queue. This time things are piling up in the queue 'aia_crtfo_in_jmsq'. Hence, we increase the number of threads on this queue as well. Also, we need to realize that the DB becomes the limiting factor, but we will address this in later tests.

Implementation

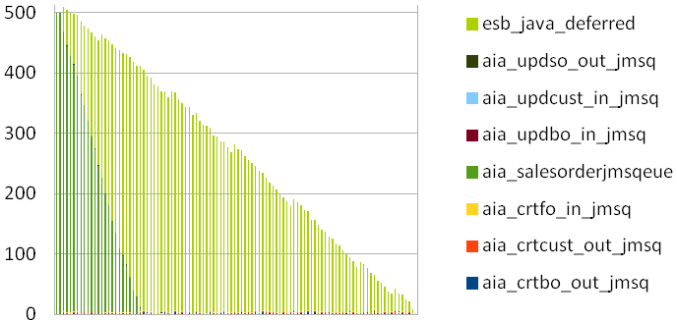
QUEUE	CONSUMER SERVICE	ADAPTER.JMS.RECEIVE.THREADS
aia_crtfo_in_jmsq	OrderOrchestrationConsumer	1 → 10

Results

From a pure throughout perspective, we do not get any further improvement by this:



But we see that we no longer queue up in any of the internal queues:



Analysis

As visualized in the queue diagram above, the number of messages waiting in queue 'aia_crtfo_in_jmsq' is no longer relevant since enough threads are picking up the incoming messages from the queue. However, overall throughput is not further increased and the overall CPU utilization is already at a fairly high level, so there is probably not much to be gained by tuning the JCA threads further.

Database Tuning

Observation: Database Host Paging

It is a best practice to run database performance statistic reports to analyze the behavior of the SOA infrastructure database. The database can easily become the bottleneck in a SOA deployment since components such as BPEL may make heavy use of the database depending on the nature of the services and their configuration.

To help you understand what is going on the database tier, the Oracle database server provides various reports such as the AWR and the ADDM reports. These reports are implemented as SQL scripts (awrrpt.sql and addmrpt.sql) and can be found in the directory <ORACLE_HOME>/rdbms/admin. See the Oracle database documentation for more details.

While both reports rely on the same set of database statistics held in the workload repository, the ADDM report highlights the most relevant areas while the AWR report provides a full list of all performance details. We review the ADDM report created for the time range of tests previously executed. The following snippet shows the first section of the ADDM report:

```
        DETAILED ADDM REPORT FOR TASK 'TASK_164' WITH ID 164
        ----------------------------------------------------

            Analysis Period: 15-DEC-2011 from 01:33:56 to 07:00:48
      Database ID/Instance: 1293781610/1
   Database/Instance Names: ORCL/orcl
                 Host Name: db.aiaperf.com
          Database Version: 10.2.0.5.0
            Snapshot Range: from 54 to 60
```

```
                Database Time: 55563 seconds
          Average Database Load: 2.8 active sessions

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

FINDING 1: 100% impact (55563 seconds)
-------------------------------------
Significant virtual memory paging was detected on the host operating system.

   RECOMMENDATION 1: Host Configuration, 100% benefit (55563 seconds)
      ACTION: Host operating system was experiencing significant paging but no
         particular root cause could be detected. Investigate processes that
         do not belong to this instance running on the host that are consuming
         significant amount of virtual memory. Also consider adding more
         physical memory to the host.
```

Analysis

It is clearly not a good situation when the database operating system needs to swap since the available physical memory is not sufficient. So there are two possible ways to improve the situation: either add more memory to the machine hosting the database, or check if there is a way to reduce memory consumption of the database itself so that paging no longer happens. To check the second option, we also use the AWR report that gives us the following SGA target advisory:

SGA TARGET SIZE (M)	SGA SIZE FACTOR	EST DB TIME (S)	EST PHYSICAL READS
292	0.25	60,255	617,906
584	0.50	56,420	381,961
876	0.75	55,991	364,315
1,168	1.00	55,812	353,635
1,460	1.25	55,672	342,884
1,752	1.50	55,499	334,327
2,044	1.75	55,466	332,452
2,336	2.00	55,466	332,452

From this, we can conclude that reducing the memory footprint of the database from the current assignment of 1.168M to the SGA to a lower value will not lead to significantly more physical reads on the disk. Therefore, we decide to reduce.

Implementation

The SGA target value can be changed in any SQL client tool in the following way:

```
SQL> alter system set sga_target=768M scope=spfile;

System altered.

SQL> alter system set sga_max_size=768M scope=spfile;
```

```
System altered.
```

Note this change requires a database restart to take effect.

Observation: Log contention

The ADDM report shows various issues related to log files. The following entries refer to log files:

```
FINDING 6: 12% impact (6918 seconds)
------------------------------------
Waits on event "log file sync" while performing COMMIT and ROLLBACK operations
were consuming significant database time.

   RECOMMENDATION 1: Host Configuration, 12% benefit (6918 seconds)
      ACTION: Investigate the possibility of improving the performance of I/O
         to the online redo log files.
      RATIONALE: The average size of writes to the online redo log files was
         128 K and the average time per write was 21 milliseconds.

   SYMPTOMS THAT LED TO THE FINDING:
      SYMPTOM: Wait class "Commit" was consuming significant database time.
               (12% impact [6918 seconds])

FINDING 7: 11% impact (5961 seconds)
------------------------------------
Log file switch operations were consuming significant database time while
waiting for checkpoint completion.

   RECOMMENDATION 1: DB Configuration, 11% benefit (5961 seconds)
      ACTION: Verify whether incremental shipping was used for standby
         databases.

   RECOMMENDATION 2: DB Configuration, 11% benefit (5961 seconds)
      ACTION: Increase the size of the log files to 1050 M to hold at least 20
         minutes of redo information.

   ADDITIONAL INFORMATION:
      This problem can be caused by use of hot backup mode on tablespaces.
      DML to tablespaces in hot backup mode causes generation of additional
      redo.

   SYMPTOMS THAT LED TO THE FINDING:
      SYMPTOM: Wait class "Configuration" was consuming significant database
               time. (27% impact [14820 seconds])
```

Analysis

Initially, the log files are co-located with the regular database files and have a default size of 50M. According to the findings in the report, we need to move the files to a dedicated disk (mounted at /mnt/second in our case), and increase the log file size so that we can assume around 3 log switches per hour which is an ideal value.

Implementation

First, the current configuration and status of the log files can be viewed using the Oracle system views V$LOG and V$LOGFILES. In this scenario, the current log file turns out to be log file 2 and, hence,

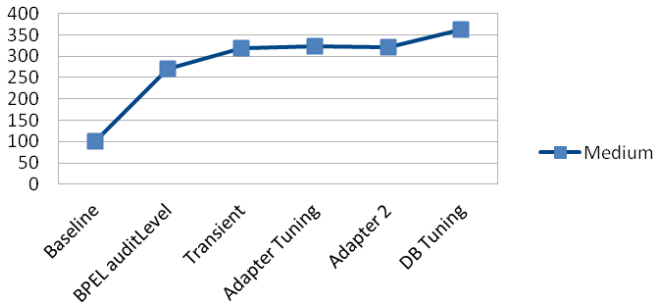
we first drop and re-create the other, then do an enforced log switch and finally also re-create the second log file :

```
ALTER DATABASE OPEN;

SELECT * from V$LOG;

ALTER DATABASE DROP LOGFILE GROUP 3;

ALTER DATABASE ADD LOGFILE GROUP 3 ('/mnt/second/oracle/redo003.log') SIZE 1024M;

ALTER DATABASE DROP LOGFILE GROUP 1;

ALTER DATABASE ADD LOGFILE GROUP 1 ('/mnt/second/oracle/redo001.log') SIZE 1024M;

ALTER SYSTEM SWITCH LOGFILE;

ALTER DATABASE DROP LOGFILE GROUP 2;

ALTER DATABASE ADD LOGFILE GROUP 2 ('/mnt/second/oracle/redo002.log') SIZE 1024M;
```

After that, the log files are located on another physical disk and have a size of 1GB.

Results

The changes on the database configuration lead to the following test results:



Observation: Missing Indexes on X-Referencing Table

The ADDM report highlights SQL statements related to the X-Referencing database table:

```
FINDING 4: 9.8% impact (586 seconds)
-----------------------------------
SQL statements consuming significant database time were found.

   RECOMMENDATION 1: SQL Tuning, 7.5% benefit (453 seconds)
      ACTION: Tune the PL/SQL block with SQL_ID "c92uxxwhs2sbz". Refer to the
         "Tuning PL/SQL Applications" chapter of Oracle's "PL/SQL User's Guide
         and Reference"
         RELEVANT OBJECT: SQL statement with SQL_ID c92uxxwhs2sbz
         begin dbms_aqin.aq$_dequeue_in( :1, :2, :3, :4, :5, :6, :7, :8, :9,
         :10, :11, :12, :13, :14, :15, :16, :17, :18, :19, :20, :21, :22, :23,
         :24, :25, :26, :27, :28, :29); end;
      RATIONALE: SQL statement with SQL_ID "c92uxxwhs2sbz" was executed 139616
         times and had an average elapsed time of 0.0065 seconds.
```

```
   RECOMMENDATION 2: SQL Tuning, 3.1% benefit (187 seconds)
      ACTION: Run SQL Tuning Advisor on the SQL statement with SQL_ID
         "212g7g504z7nm".
         RELEVANT OBJECT: SQL statement with SQL_ID 212g7g504z7nm and
         PLAN_HASH 2435635183
         select VALUE from XREF_DATA  where XREF_COLUMN_NAME = :1  and
         XREF_TABLE_NAME = :2  and ROW_NUMBER in  ( select ROW_NUMBER from
         XREF_DATA    where XREF_COLUMN_NAME = :3    and XREF_TABLE_NAME = :4
         and VALUE = :5    and IS_DELETED = 'N' )   and IS_DELETED = 'N'
      ACTION: Investigate the SQL statement with SQL_ID "212g7g504z7nm" for
         possible performance improvements.
         RELEVANT OBJECT: SQL statement with SQL_ID 212g7g504z7nm and
         PLAN_HASH 2435635183
         select VALUE from XREF_DATA  where XREF_COLUMN_NAME = :1  and
         XREF_TABLE_NAME = :2  and ROW_NUMBER in  ( select ROW_NUMBER from
         XREF_DATA    where XREF_COLUMN_NAME = :3    and XREF_TABLE_NAME = :4
         and VALUE = :5    and IS_DELETED = 'N' )   and IS_DELETED = 'N'
      RATIONALE: SQL statement with SQL_ID "212g7g504z7nm" was executed 229419
         times and had an average elapsed time of 0.00058 seconds.
```

Analysis

The ADDM report highlights that access to the X-Referencing table consumes significant database server resources. Since the queries themselves are part of the Fusion Middleware stack, there is no way to directly change the queries. But we can still 'help' the database to better perform for this sort of queries by providing additional indexes.

Note: Identifying the right set of indexes to provide the best performance is a non-trivial task requiring an experienced database administrator. Certainly it is not true that more indexes necessarily give a better index. And it is also not true that an index that worked on one environment has the same impact on another, since there are many influencing factors that need to be accounted for.

One of the key prerequisites for Databases to perform well is to supply them with current statistics so that the query optimizer is able to come up with good execution plans leading to good database performance. There are many ways to gather statistics in an Oracle Database. In our case, we execute the dbms_stats.gather_database_stats procedure ahead of every test run in order to start tests with current statistics.

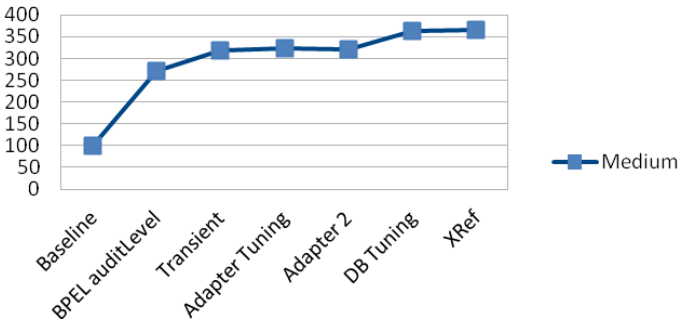
Implementation

Add the following index to improve performance for database queries against the AIA X-Referencing database table xref_data:

```
CREATE INDEX "AIA"."XREF_TABLE_COL_ROW" ON "AIA"."XREF_DATA"
  (
    "XREF_TABLE_NAME",
    "XREF_COLUMN_NAME",
    "ROW_NUMBER"
  )
  [optional STORAGE_CLAUSE]
 ;
```

Results

The test results now look like this:



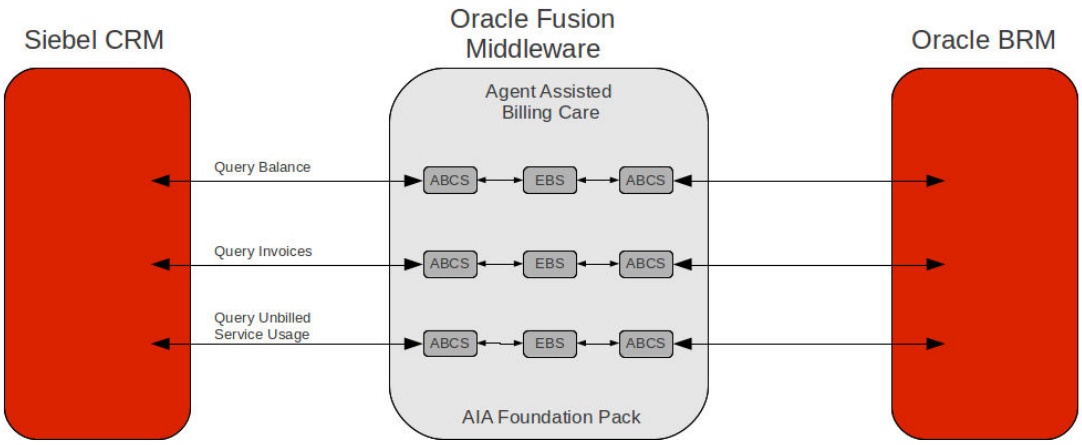
We stop the tuning exercise for the asynchronous flow at this point since the last tuning attempts have shown no significant improvements and we do not observe anything major on both tiers that would indicate larger tuning opportunities. The last number of 365 indicates that we have improved the throughput around 3.5 times as compared to the initial, not-tuned configuration.

We will take a closer look on tuning a set of synchronous flows.

Synchronous Integration Flows

Synchronous Test Scenario

For the synchronous tests, we use various online queries from the AIA Agent Assisted Billing Care PIP. The integration flows provided by this PIP are all triggered by a user in the Siebel CRM application by navigating to some particular screens that are part of the AIA PIP. In order to display information from the billing application (BRM), Siebel calls the middleware via various SOAP calls towards AIA connector services (ABCS).



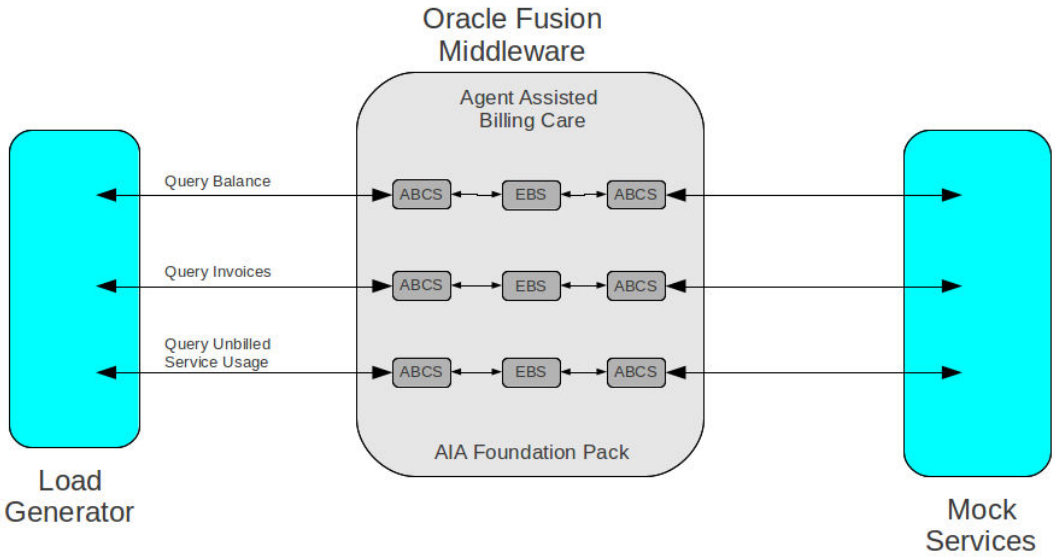
Similar to the asynchronous flow, we simulate BRM by introducing the required mock services that play the role of the BRM APIs for our tests.

To simulate a mix of concurrent synchronous requests for various accounts originating from Siebel, we use Apache JMeter to initiate concurrent SOAP calls for the following flows (this is only a subset of flows supported by the PIP):

- Query account balance
- Query invoices
- Query unbilled service usage

Testing Methodology

The testing methodology for the synchronous case is quite different from the asynchronous test because we now need to simulate concurrent SOAP requests reaching AIA.



We configure three different SOAP request messages matching the three flows mentioned above. We keep these requests dynamic by executing the queries for a set of a 1000 different customers.

We configure JMeter to run these queries by 20 parallel threads in a continuous fashion (500 requests per thread per query) generating 30.000 requests overall for a single test.

Measuring Performance

The main performance indicator for us is still throughput. In addition to that, we also monitor minimum, maximum, and average response times across all requests every time we run a test of 30,000 requests.

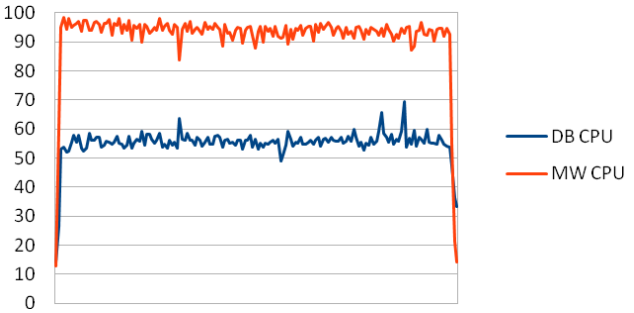
This time we do not get this information from the BPEL instance table CUBE_INSTANCE because we are going to make all BPEL processes transient later and therefore will not be able to see anything there. Instead JMeter can provide all the performance numbers from the usual JMeter screens.

Baseline Test Results

In our initial test we see the following values:

INDICATOR	VALUE
Throughput	764 requests per minute
Min. Response Time	0.146 seconds
Max. Response Time	6.498 seconds
Avg. Response Time	1.460 seconds

Note that these initial numbers already benefit from all the tuning activities that have been done for the asynchronous flows, e.g. database tuning.



We continue to observe CPU utilization on both tiers. The synchronous use case initially creates quite constant full utilization on the SOA tier and a 55% CPU utilization on the database tier:

Turning Off Dehydration

Observation

Simulating 20 concurrent users constantly sending synchronous requests to the middleware shows fully utilized CPU resources on the middleware tier. Nevertheless, the database node also has a high utilization given the nature of the requests.

Analysis

The only database related requests for this kind of synchronous flows are lookups against the X-Referencing table and the persistence of the service instances. Therefore, the first change we make is to

change all six used BPEL services to act as transient services, so they no longer store the services instances in the database except in the case of an error.

Implementation

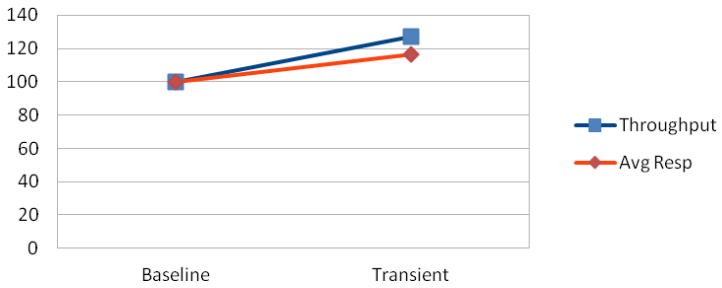
Refer to the asynchronous test case for implementation details. For the AIA Agent Assisted Billing Care PIP, all used BPEL services qualify as transient services and we apply this to the following services:

- QueryBalanceSummarySiebelCommsReqABCSImpl
- QueryInvoiceListSiebelCommsReqABCSImpl
- QueryUnbilledUsageSiebelCommsReqABCSImpl
- QueryCustomerPartyListBRMCommsProvABCSImpl
- QueryInvoiceListBRMCommsProvABCSImpl
- QueryServiceUsageBRMCommsProvABCSImpl

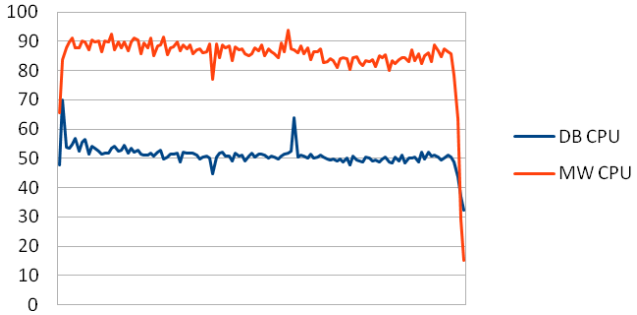
Note: This list only covers the test cases in our example. The AABC PIP supports more flows, so in a customer deployment the list of relevant services for this kind of tuning would include more services.

Results

Changing to transient BPEL services improves throughput by 27% and the average response time by 16%:



The system CPU visualization shows that the average CPU load on the database tier is down on both database server and application server tier by an average of 5-10%:



Missing Performance Patch

Observation

The ADDM report shows the following results:

```
FINDING 2: 17% impact (457 seconds)
----------------------------------
SQL statements consuming significant database time were found.

   RECOMMENDATION 1: SQL Tuning, 12% benefit (309 seconds)
      ACTION: Investigate the SQL statement with SQL_ID "4x3mnmcq9mkk7" for
         possible performance improvements.
         RELEVANT OBJECT: SQL statement with SQL_ID 4x3mnmcq9mkk7 and
         PLAN_HASH 2435971774
         SELECT ID,SERVICE_GUID,OPERATION_GUID FROM ESB_RELATION_XML WHERE
         IS_STALE='N'
      RATIONALE: SQL statement with SQL_ID "4x3mnmcq9mkk7" was executed 5134
         times and had an average elapsed time of 0.03 seconds.
```

Analysis

It is a good practice to check the resources at My Oracle Support when you see suspicious SQL statements indicating weak performance. In this case, we refer to note 1352321.1 which suggests applying patch 9896525.

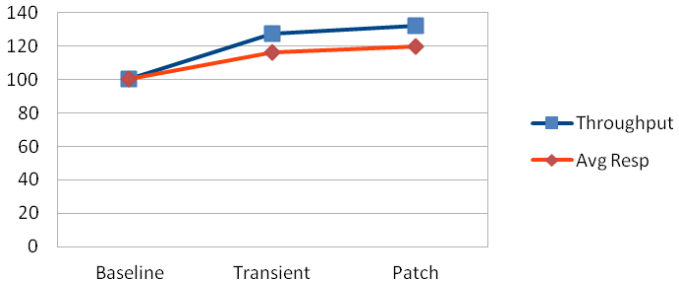
Even though the query shown in the note does not exactly match the finding of the ADDM report (however it does refer to the same database table ESB_RELATION_XML), we still want to give it a try.

Implementation

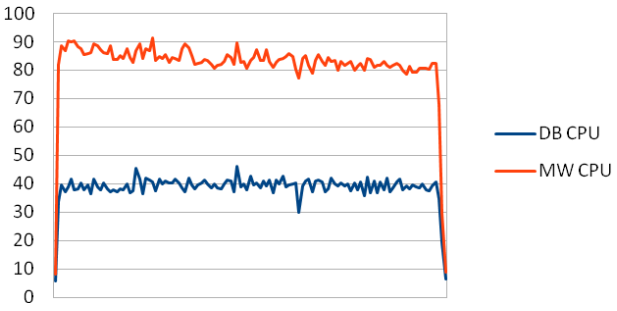
The patch can be applied to all nodes of the AIA cluster using the usual Opatch tool.

Results

The finding shown above no longer shows up in the database reports and the application of the patch leads to a further improvement of throughput by 5%:



The system CPU load is visualized below. It indicates that the patch leads to a reduction in database server load of about 10% in average when compared to the plot of the previous test:



Turning off ESB Instance Tracking

Observation

In the same way that we made BPEL transient in order to relieve the system from storing instance data, we can use a similar switch for the ESB engine to avoid the storage of ESB instance data.

Analysis

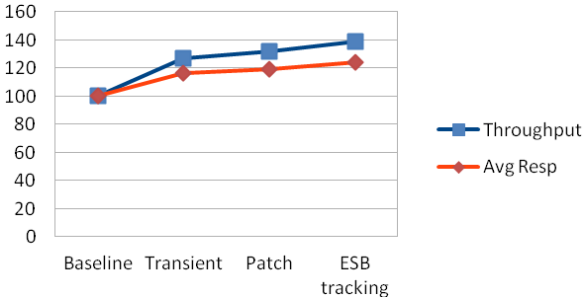
In ESB this configuration is a global setting (and not per service as in the case of BPEL) and therefore might not be applicable in every case. The use case as well as error handling and monitoring requirements will determine if this is possible or not. In the case of purely synchronous flows, there is a good chance that this can be used since faults are always covered by the service consumer (Siebel in our case) anyway and there is no point in persisting the ESB instance data even more since the faulted transient BPEL instances will be persisted in the case of an error.

Implementation

ESB instance tracking can be turned off in the ESB console by clearing the 'Enable tracking of instances' checkbox which is checked by default:

Results

Turning off the storage of every ESB instance in the infrastructure database leads to another gain of 5-7% in our case:



The system CPU load is visualized below. The change leads to a even lower utilization of the database tier which was expected since we do not store any instance details for BPEL or ESB services at this point.

The remaining database load should mainly be related to X-Referencing lookups that are executed by the synchronous BPEL services.

The Impact of External Application Latency

Observation

From the previous test run we can see that even though there is very low dependency on the database tier, the application server tier's CPU is still not fully utilized.

Analysis

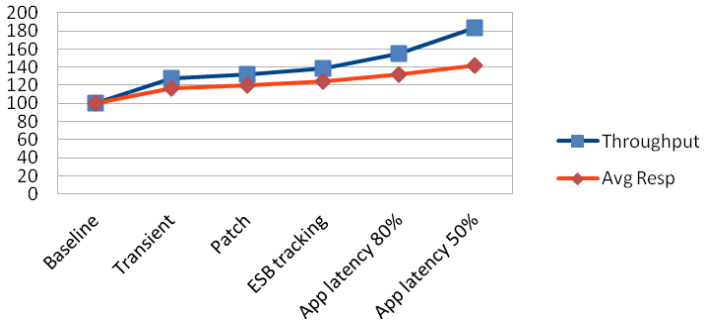
This effect is due to the middleware waiting for an external application in a blocked wait mode, that is the middleware threads cannot do anything other than wait for the response from the application. In our case, we have intentionally implemented certain delays in the mock services simulating the BRM system to get a realistic behavior of the simulated application. Now we want to find out how improving the response times of the external system will improve the overall performance. With the help of the mock services we are able to do a what-if analysis to figure out if tuning the external application would be beneficial.

Implementation

The current average response time for the BRM mock services is configured to be 500 milliseconds. For this test, we change the SOAP UI mock services to respond in 400 (i.e. latency down to 80% of the original value) and 250 (50%) milliseconds on average. In other words, the results of this test will show what overall throughput could be achieved if the BRM system was tuned to provide better response times.

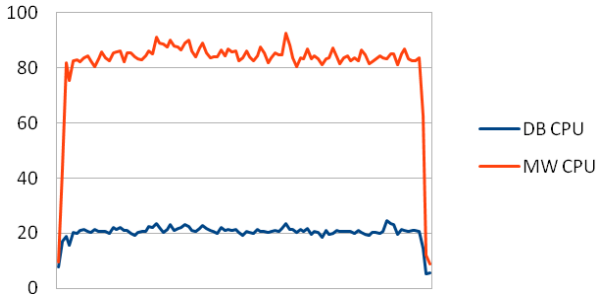
Results

The lower external latency leads to an improved throughput by 15% (for 80% latency) and 45% (for 50% latency):

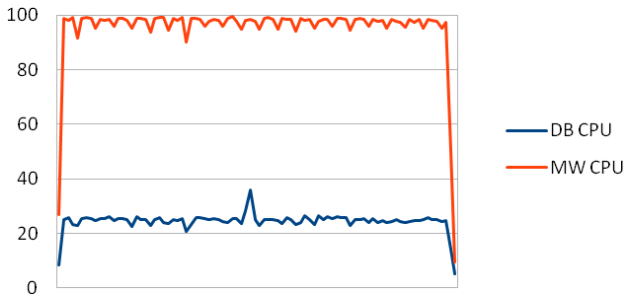


The system CPU load is visualized below for both runs. It can be clearly seen that less waiting for the external application leads to a better / full utilization of the middleware CPU resources.

CPU utilization for 80% application latency:



CPU utilization for 50% application latency:



Additional Tuning Strategies

There are several other fundamental strategies that could improve performance significantly that were not covered in this guide. This section discusses additional options for you to consider in other tuning projects.

Horizontal Scaling

One of the most obvious ways to increase the system's capacity is to add more servers to an existing cluster setup. While this has proven to work very well with Oracle Fusion Middleware and Oracle RAC databases, it is the wrong assumption that double hardware will automatically lead to effectively double capacity or double performance. You must be aware that clusters come with a certain degree of overhead for managing the cluster infrastructure. For example, adding more application server nodes to a cluster automatically increases the load on the database tier and you must make sure the load can be handled well on all tiers. The overall system performance can never be better than whatever the weakest piece in the mix is able to do.

Vertical Scaling

While you add more hardware in the case of horizontal scaling, the vertical scaling goes into a different direction. Here the goal is to better utilize the available hardware resources. Imagine the middleware server runs on huge boxes having lots of CPUs and plenty of memory. In such a case, a single SOA

10g server would not be able to leverage all the power that the hardware actually offers. The way to improve this would be to use the same box to run multiple nodes of the overall SOA cluster on the same hardware. This is possible in 10g when you work carefully to avoid clashes of the network ports used by each cluster node, but it will require separate binary installations per node. There is a certain installation effort required, but that could quickly pay off by much better utilization of the available hardware resources.

Specialized Deployment

In this guide we have seen that synchronous and asynchronous flows do have different demands against the underlying SOA infrastructure. This could lead to a situation where you tune the SOA infrastructure in a way to serve both types of flows as effectively as possible. But it is clear that it would be a compromise and it would not be optimal for either one of the flow types.

So if such a compromise is not acceptable, there is always the option of setting up multiple 'specialized' clusters, that is,. a cluster hosting synchronous integration flows and another one hosting asynchronous integration flows. In the example of this guide, that would mean deploying the Order-to-Bill PIP to one cluster, and the Agent Assisted Billing Care PIP to the other. That would allow tuning of both clusters independently. However, it is important to state that there are some things to consider in such a topology. For example, both clusters would have to share the same X-Referencing database table. This can be achieved by configuring the middleware data sources accordingly.

White Paper Title
April 2012
Author: Gerhard Dresch

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together